# Review on Control structures

Like C, C++ also supports the following three control structures

1. Sequential structure (straight line)

2. Selection structure (branching or decision making)

3. Looping structure (iteration or repetition)

# Functions in C++

## 3. 1 Introduction

A function is a named unit of a group of program statements. The unit can be invoked from other part of the program. Dividing a program into function is one of the major principles of top-down structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

void show( ) ; **/ \* Function declaration \* /**

int main ( )

{

- - - - - -

- - - - - -

show( ) ; **/ \* Function call \* /**

}

void show( ) **/ \* Function definition \* /**

{

- - - - - -

- - - - - - **/ \* Function body \* /**

- - - - - -

}

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and controls return to the main program when the closing brace is encountered. C++ has added many new features to functions to make them more reliable and flexible.

## 3. 2 C++ User-defined Function Types

1. **Function with no argument and no return value**

#include<iostream>

using namespace std;

void msg() //eliminating the function declaration(function definition preceding function call).

{

cout<<" true beauty ";

}

int main()

{

msg(); //function call.

return 0;

}

1. **Function with no argument but return value**

* #include<iostream>

using namespace std;

int area();

int main()

{

cout<<"Area: "<<area();

return 0;

}

int area()

{

int l, b;

cout<< "Enter l, and b: "<<endl;

cin>>l>>b;

return l\*b;

}

1. **Function with argument but no return value**

* void area(int, int); //function declaration.

1. **Function with argument and return value.**

* float area(float, int); //function declaration.

## 3.3 Passing Arguments to Function

An argument is a data passed from a program to the function. In function, we can pass a variable by three ways:

1. **Passing by value**

In this the value of actual parameter is passed to formal parameter when we call the function. But actual parameters are not changed.

#include<iostream>

using namespace std;

void swap(int, int) ;

int main ( )

{

void swap(int a, int b) // function definition

{

int t ;

t=a ;

a=b ;

b=t ;

}

**Output:**

x = 10

y = 20

int x, y ;

x=10 ;

y=20 ;

swap(x, y) ;

cout << "x = "<<x<<endl ;

cout<< "y = " <<y<<endl ;

return 0;

}

**2. Passing by reference**

Passing argument by reference uses a different approach. In this, the reference of original variable is passed to function. Formal parameters become references (aliases) to the actual parameters in the calling function. The main advantage of passing by reference is that the function can access the actual variable in the calling program. The second advantage is this provides a mechanism for returning more than one value from the function back to the calling program.

**Example:**

void swap(int &, int &) ; //reference variables as arguments.

int main ( )

{

int x, y ;

x=10 ;

y=20 ;

swap(x, y) ; //Function call.

cout << "x = "<<x<<endl ;

cout<< "y = " <<y<<endl ;

return 0;

}

void swap(int & a, int & b) // function definition.

{

int t ;

t=a ;

a=b ;

b=t ;

}

**Output:**

x=20

y=10

**3. Passing by address or pointer**

This is similar to passing by reference but only difference is in this case, we can pass the address of a variable.

#include<iostream>

void swap(int \*a, int \*b) // function definition

{

int t ;

t= \*a;

\*a= \*b;

\*b= t;

}

**Output:**

x=20

y=10

using namespace std;

void swap(int \*, int \*) ;

int main ( )

{

int x, y ;

x=10 ;

y=20 ;

swap(&x, &y) ; //Passing addresses of actual parameters.

cout << "x = "<<x<<endl ;

cout<< "y = " <<y<<endl ;

return 0;

}

## 3.4 Function Overloading

Function overloading (also method overloading) is a programming concept that allows programmers to define two or more functions with the same name and in the same scope.

Each function has a unique signature (or header), which is derived from:

* function/procedure name
* number of arguments
* arguments' type
* arguments' order
* arguments' name

Please note: Not all above signature options are available in all programming languages.

Two or more functions can share the same name as long as either the type of their arguments differs or the number of their arguments differs or both. When two more functions share the same name but with different parameters, different number of parameters, or both, they are said overloaded.

Overloaded functions can help reduce the complexity of a program by allowing related operations to be referred to by the same name.

To overload a function, simply declare and define all required versions. The compiler will automatically select the correct version based upon the number and / or type of arguments used to call the function.

**// Program illustrate function overloading.** // Function area( ) is overloaded.

#include <iostream>

int area(int) ;

double area(double, int) ; **// Declarations (prototypes)**

long area(long, int, int) ;

int main( )

{

cout<<area(10)<< endl ;

cout<<area(2.5,8)<< endl ;

cout<<area(100, 75, 15)<< endl ;

return 0 ;

}

int area(int s) **// square**

{

return(s\*s) ;

**Output:**

100

125.6

20250

}

double area(double r, int h) **// Surface area of cylinder ;**

{

return(2\*3.14\*r\*h) ;

}

long area(long l, int b, int h) **//area of parallelepiped**

{

return(2\*(l\*b+b\*h+l\*h)) ;

}

## 3.5 Default Arguments

In C++ programming, you can provide default values for function parameters. The idea behind default argument is simple. If a function is called by passing argument/s, those arguments are used by the function. But if the argument/s are not passed while invoking a function then, the default values are used. Default value/s are passed to argument/s in the function prototype.

C++ allows us to assign default value(s) to a function’s parameters, which are called default arguments. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. *If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified, this default value is ignored and the passed value is used instead.*

**Example:**

void f(int i, int j=7) ; // legal

void g(int i=3, int j) ; // illegal

void h(int i, int j=3,int k=7) ; // legal

void m(int i=1, int j=2, int k=3) ; // legal

void n(int i=2, int j, int k=3) ; // illegal, why**?**

**Remember:** *Only the trailing arguments can have default values i.e., that is we must add default values from right to left.*

**SAMPLE PROGRAM:**

#include <iostream>

using namespace std;

int divide (int , int =2); **// function declaration; with one default argument.**

int main ()

{

cout << divide (12); **//default for 2nd argument.**

cout << endl;

cout << divide (20,4); **//pass all arguments explicitly.**

return 0;

}

int divide (int a, int b) **//function definition**

{

int r;

**Note:**

*The default values are specified at the time of function declaration.*

*Any argument in a function cannot have a default value unless all arguments appearing on its right have their default values.*

*If a function has N default arguments then it can be invoked N+1 different ways.*

r=a/b;

return (r);

}

**Another Way:**

#include<iostream>

using namespace std;

int divide (int a , int f=2) **// function prototype; with one default argument.**

{

return a/f;

}

int main ()

{

cout << divide (12); **//default for 2nd argument.**

cout << endl;

cout << divide (20, 4); **//pass all arguments explicitly.**

return 0;

}

**Advantages of providing default arguments:**

1. Default arguments are useful when some arguments always have the same value.
2. We can use default arguments to add new parameters to the existing functions.
3. Default arguments can be used to combine similar functions into one.

## 3.6 Inline Functions

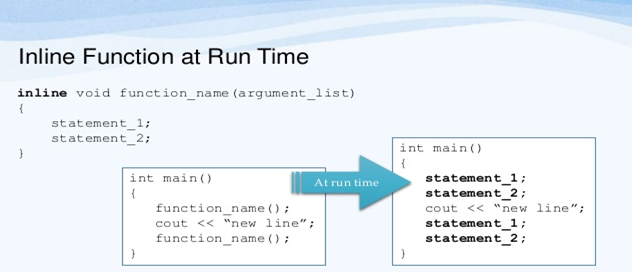
Function call is a costly operation. During the function call it’s execution take overheads like: Saving the values of registers, Saving the return address, Pushing arguments in the stack, Jumping to the called function, Loading registers with new values, Returning to the calling function, and reloading the registers with previously stored values. For large functions this overhead is negligible but for small function taking such large overhead is not justifiable. To solve this problem concept of *inline function* is introduced in C++.

The inline functions are a C++ enhancement feature to increase the execution time of a program. Inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function body.

Inline functions are similar to macros, but macros are not functions. And macros are treated differently by the compiler.

To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

The compiler replaces the function call statement with the function code itself (process called expansion) and then compiles the entire code. Thus, with inline functions, the compiler does not have to jump to another location to execute the function, and then jump back as the code of the called function is already available to the calling program.



The advantage of inline functions is that they can be executed much faster than normal functions. The disadvantage of inline functions is that it takes more memory than normal function. In general, only short functions are declared as inline functions.

To in-line a function, simply precede the function’s definition with the inline keyword.

**Syntax:**

*inline* return\_type function\_name(arguments)

{

**//function body.**

}

int main()

{

int x, y, s;

cout<<”Enter two numbers:”<<endl;

cin>>x>>y

s=sum(x, y); **//function call.**

cout<< “Sum= ”<<s<<endl;

}

**EXAMPLE:**

#include <iostream>

using namespace std;

inline int sum(int a, int b)

{

return a+b;

}

*Here at the time of function call instead of jumping to the called function, function call statement is replaced by the body of the function.* So there is no function call overhead.

### Pros:-

1. It speeds up your program by avoiding function calling overhead.

2. It save overhead of variables push/pop on the stack, when function calling happens.

3. It save overhead of return call from a function.

4. It increases locality of reference by utilizing instruction cache.

5. By marking it as inline, you can put a function definition in a header file (i.e. it can be included in multiple compilation unit, without the linker complaining)

### Cons:-

1. It increases the executable size due to code expansion.

2. C++ inlining is resolved at compile time. Which means if you change the code of the inlined function, you would need to recompile all the code using it to make sure it will be updated

3. When used in a header, it makes your header file larger with information which users don’t care.

## 3.7 Math library functions

Functions come in two varieties. They can be defined by the user or built in as part of the compiler package. As we have seen, user-defined functions have to be declared at the top of the file. Built-in functions, however, are declared in **header files** using the #include directive at the top of the program file, e.g. for common mathematical calculations we include the file cmath with the #include<cmath> directive which contains the *function prototypes* for the mathematical functions in the cmath library.

### Mathematical functions

Math library functions allow the programmer to perform a number of common mathematical calculations:

|  |  |
| --- | --- |
| ***Function*** | ***Description*** |
| sqrt(x) | square root |
| sin(x) | trigonometric sine of x (in radians) |
| cos(x) | trigonometric cosine of x (in radians) |
| tan(x) | trigonometric tangent of x (in radians) |
| exp(x) | exponential function |
| log(x) | natural logarithm of x (base e) |
| log10(x) | logarithm of x to base 10 |
| fabs(x) | absolute value (unsigned) |
| ceil(x) | rounds x up to nearest integer |
| floor(x) | rounds x down to nearest integer |
| pow(x, y) | x raised to power y |

#### Sample Program:

#include<iostream>

#include<cmath>

using namespace std;

int main()

{

int a,b, c;

float d=45.6,f=78.9;

cout<<"enter values of a & b: " <<endl;

cin>>a>>b;

c = pow(a, b);

cout<<b<<" Power of "<<a<<": "<<c<<endl;

cout<<"Square root of "<<a<<": "<<sqrt(a)<<endl;;

cout<<"Ceiling of "<<d<<": "<<ceil(d)<<endl;

cout<<"Floor of "<<d<<": "<<floor(d)<<endl;

return 0; }

## 3.8 Storage Classes

* A storage class defines the ***scope*** (visibility) and ***life-time*** of variables and/or functions within a C++ Program. How storage is allocated for variables and how variable is treated by complier depends on these storage classes.
* The scope of the variable determines which parts of the program can access it.
* And the lifetime refers how long it stays in existence. That is, the duration till which a variables remains active during program execution.
* There are four storage classes:
  + **auto**
  + **register**
  + **static**
  + **extern or global**

### 3.8.1 The auto Storage Class

* The **auto** storage class is the default storage class for all local variables.
* Its visibility is restricted to the function in which it is declared. Its lifetime is also limited to till the time its container function (function in which they are declared) executing.
* **Example:**

void somefunc()

{

int count; ***//by default auto***

auto int month;

}

* The example above defines two variables with the same storage class; auto can only be used within functions, i.e., local variables.

### 3.8.2 The register Storage Class

* Similar in behavior(*visibility & lifetime*) to an automatic variable, except how they are stored in the memory. automatic variables are stored in the primary memory but register variables are stored in CPU register.
* The objective of the register variable is to increase the access speed to execute the program faster.
* **Example:**

{

register int length;

}

### 3.8.3 The extern Storage Class

* Variables that are declared outside of any function are called External or global variables. These variables have external storage class.
* A global variables are visible to all the functions present in the program i.e., global scope.
* The lifetime of a global or external variable is same as the lifetime of a program.
* **Example:**

int a=20; // global or external variable

void somefunc()

{

int x;

x = a+20;

}

### 3.8.4 The static Storage Class

* Variables which are declared using the keyword static are called static variables.
* A static variable has the visibility of a local variable but the lifetime of the external variable.
* **Example:**

void func(void);

static int count = 15; **/\* Global static variable \*/**

int main()

{

do

{

func();

count--;

} while(count > 10);

return 0;

}

void func( void ) **// Function definition**

{

static int i = 5; **// local static variable**

i++;

cout << "i is " << i ;

cout << " and count is " << count << endl;

}

### Summary:

### 

### Important Questions

1. Discuss the feature of Object-Oriented Programming? Differentiate between Object-Oriented Programming & Procedural Based Programming.
2. What is type casting? Explain with suitable example.
3. Write a program to find the cube of given integer using inline function.
4. Write a function using reference variables as arguments to swap the values of a pair of integers.
5. Write a program that inputs a character from keyboard and displays its corresponding ASCII value on the screen.
6. What is function overloading? Explain with suitable example.
7. Write a program to compute the area of a triangle, a rectangle, square, and a circle by overloading the area() function.
8. Explain the inline function with example.
9. WAP to find the square of given integer using inline function.
10. Write a program to find the square root of given integer using inline function.
11. Explain the use of default arguments with suitable example.
12. Explain the do/while structure.
13. Explain the use of break and continue statements in switch case statements in C++
14. Differentiate between function and macro with suitable examples.
15. Differentiate between user defined functions and library functions with suitable examples.
16. Explain different storage classes of C++.